ELSEVIER

Science of Computer Programming 27 (1996) 263–288

Science of Computer Programming

Memoizing purely functional top-down backtracking language processors

Richard A. Frost*, Barbara Szydlowski

School of Computer Science, University of Windsor, Windsor, Ont., Canada N9B 3P4

Received 1 April 1995; revised 1 April 1996
Communicated by R. Bird

Abstract

Language processors may be implemented directly as functions. In a programming language that supports higher-order functions, large processors can be built by combining smaller components using higher-order functions corresponding to alternation and sequencing in the BNF notation of the grammar of the language to be processed. If the higher-order functions are defined to implement a top-down backtracking parsing strategy, the processors are modular and, owing to the fact that they resemble BNF notation, are easy to understand and modify. A major disadvantage of this approach is that the resulting processors have exponential time and space complexity in the worst case owing to the reapplication of processors during backtracking. This paper shows how a technique called memoization can be used to improve the efficiency of such processors whilst preserving their modularity. We show that memoized functional recognizers constructed for arbitrary non-left-recursive grammars have $O(n^3)$ complexity where n is the length of the input to be processed. The paper also shows how the initial processors could have been memoized using a monadic approach, and discusses the advantages of reengineering the definitions in this way.

1. Constructing modular non-deterministic language processors in functional programming languages

One approach to implementing language processors in a modern functional programming language is to define a number of higher-order functions which when used as infix operators (denoted in this paper by the prefix $) enable processors to be built with structures that have a direct correspondence to the grammars defining the languages to be processed. For example, the function s, defined in the functional program in Fig. 1, is a recognizer for the language defined by the grammar `s ::= 'a' s s|empty` if the functions `term`, `orelse`, `then` and `empty` are defined as shown in the next few pages of this paper.

* Corresponding author. E-mail: richard@cs.uwindsor.ca.

PII S0167-6423(96)00014-7

```
s   =  (a $then s $then s) $orelse empty

a   =  term  'a'
```

Fig. 1. A functional program containing a definition of the recognizer s.

This approach, which is described in detail in Hutton [10], was originally proposed by Burge [2] and further developed by Wadler [18] and Fairburn [4]. It is now frequently used by the functional-programming community for language prototyping and natural-language processing. In the following, we describe the approach with respect to language recognizers although the technique can be readily extended to parsers, syntax-directed evaluators and executable specifications of attribute grammars [6, 7, 1, 12].

According to the approach, recognizers are functions mapping lists of inputs to lists of outputs. Each entry in the input list is a sequence of tokens to be analyzed. Each entry in the output list is a sequence of tokens yet to be processed. Using the notion of "failure as a list of successes" [18] an empty output list signifies that a recognizer has failed to recognize the input. Multiple entries in the output occur when the input is ambiguous. In the examples in this paper it is assumed that all tokens are single characters. The notation of the programming language Miranda[1] [17] is used throughout, rather than a functional pseudo-code, in order that readers can experiment with the definitions directly.

The types `token` and `recognizer` may be defined as follows where == means "is a synonym for", `x -> y` denotes the type of functions from objects of type `x` to objects of type `y`, and square brackets denote a list.

```
token == char

recognizer == [[token]] -> [[token]]
```

That is, a recognizer takes a list of lists of tokens as input and returns a list of lists of tokens as result. Note that this differs from the type found in many other papers on functional recognizers. The reason for this difference is that it simplifies the memoization process as will be explained later.

The simplest type of recognizer is one that recognizes a single token at the beginning of a sequence of tokens. Such recognizers may be constructed using the higher-order function `term` defined below. The notation `x :: y` declares `x` to be of type `y`. The function `concat` takes a list of lists as input and concatenates the sublists to form a single list. `map` is a higher-order function which takes a function and a list as input and returns a list that is obtained by applying the function to each element in the input list. Function application is denoted by juxtaposition, i.e. `f x` means `f` applied to `x`. Function application has higher precedence than any operator, and round brackets are used for grouping. The empty list is denoted by `[]` and the notation `x : y` denotes the

[1] Miranda is a trademark of Research Software Ltd.

list obtained by adding the element x to the front of the list y. The applicable equation is chosen through pattern matching on the left-hand side in order from top to bottom, together with the use of guards following the keyword `if`.

```
term :: token -> recognizer

term c inputs = (concat . map test_for_c) inputs
                where
                test_for_c [] = []
                test_for_c (t:ts) = [ts], if t = c
                test_for_c (t:ts) = [] , if t ~= c
```

The following illustrates the use of `term` in the construction of two recognizers `c` and `d`, and the subsequent application of these recognizers to three inputs. The notation `x => y` is to be read as " y is the result of evaluating the expression x". The empty list in the second example signifies that `c` failed to recognize a token `'c'` at the beginning of the input `''xyz''`. The notation `''x1 .. xn''` is shorthand for `['x1', .. ,'xn']`

```
c = term 'c'
d = term 'd'

c [''cxyz'']           => [''xyz'']
c [''xyz'']            => [ ]
d [''dabc'', ''dxyz''] => [''abc'', ''xyz'']
```

Alternate recognizers may be built using the higher-order function `orelse` defined below. The operator `++` appends two lists.

```
orelse :: recognizer -> recognizer -> recognizer
(p $orelse q) inputs = p inputs ++ q inputs
```

According to this definition, when a recognizer `p $orelse q` is applied to a list of inputs `inputs`, the value returned is computed by appending the results returned by the separate application of `p` to `inputs` and `q` to `inputs`. The following illustrates use of `orelse` in the construction of a recognizer `c_or_d` and the subsequent application of this recognizer to three inputs.

```
c_or_d = c $orelse d

c_or_d [''abc'']  => []
c_or_d [''cxyz''] => [''xyz'']
c_or_d [''dxyz''] => [''xyz'']
```

Sequencing of recognizers is obtained through the use of the higher-order function then defined as follows:

```
then :: recognizer -> recognizer -> recognizer

(p $then q) inputs = [] , if r = []
                   = q r , otherwise
                     where
                     r = p inputs
```

According to this definition, when a recognizer p $then q is applied to a list of inputs inputs, the result returned is an empty list if p fails when applied to inputs, otherwise the result is obtained by applying q to the result returned by p. (Note that, in general, then does not have the same effect as reverse composition. In particular, replacing p $then q by q . p will result in non-terminating computations for certain kinds of recursively defined recognizers.) The following illustrates use of then in the construction of a recognizer c_then_d, and the subsequent application of c_then_d to two inputs:

```
c_then_d = c $then d

c_then_d [''cdxy''] => [''xy'']
c_then_d [''cxyz''] => []
```

The "empty" recognizer, which always succeeds and which returns the complete list of inputs as output to be processed, is implemented as the identity function:

```
empty inputs = inputs
```

The functions term, orelse, then, and empty as defined above may be used to construct recognizers whose definitions have a direct structural relationship with the context-free grammars of the languages to be recognized. Fig. 2 illustrates this relationship.

The example application given below illustrates the use of the recognizer s and shows that the prefixes of the input ''aaa'' can be successfully recognized in nine

```
BNF grammar of the language          The program

s ::= 'a' s s | empty          s  = (a $then s $then s) $orelse empty
terminals = {'a'}              a  = term  'a'
```

Fig. 2. The relationship between the grammar and the program implementing the recognizer.

different ways. Empty strings in the output, denoted by ''', correspond to cases where the whole input ''aaa'' has been recognized as an s. The output shows that there are five ways in which this can happen. The two strings in the output consisting of ''a'' correspond to cases where the prefix''aa'' has been recognized leaving ''a'' for subsequent processing. The output shows that there are two ways in which this can happen. The string in the output consisting of two letters ''aa'' corresponds to the case where the prefix''a'' has been recognized leaving ''aa'' for subsequent processing. This can only happen in one way when ''a'' is recognized as an s.

```
s[''aaa''] => [''',''',''',''a'',''',''',''a'',''aa'',''aaa'']
```

The major advantage of this approach is that the processors created are modular executable specifications of the languages to be processed. Components can be defined, compiled and executed directly. For example, (a $then s $then s) is a recognizer that may be executed directly as for example:

```
(a $then s $then s) [''bcd''] => [ ]
(a $then s $then s) [''aab''] => [''b'', ''b'', ''ab'']
```

The advantages of building language processors using this technique come at a price. The processors employ a naive top-down fully backtracking search strategy and consequently exhibit exponential-time and space behavior in the worst case. In the following, we show how this problem can be overcome through a process of memoization. We begin by discussing techniques that have been proposed by other researchers concerning the use of memoization with top-down backtracking language processors. We then describe how memoization can be achieved at the source-code level in purely functional programming languages and show how the technique can be adapted for use to improve the efficiency of top-down backtracking recognizers. We provide a formal description of the algorithm and a proof of the complexity result. In addition, we show how the same result can be obtained in a more structured way by use of a monad. We conclude with a discussion of how the approach can be used with parsers and executable attribute grammars.

2. Memoizing language processors

Memoization [14, 9] involves a process by which functions are made to automatically recall previously computed results. Conventional implementations involve the maintenance of memo-tables which store previously computed results. Application of a memoized function begins by reference to its memo-table. If the input has been processed before, the previously computed result is returned. If the input has not been processed before, the result is computed using the original definition of the function, with the exception that all recursive calls use the memoized version, the memo-table is then updated and the result returned.

Many of the efficient algorithms for recognition and parsing make use of some kind of table to store well-formed substrings of the input and employ a form of memoization. Earley's algorithm [3] is an example. In most of these algorithms, the parsing and table update and lookup are intertwined. This results in relatively complex processors that are not modular. Norvig [16] has shown how memoization can be used to obtain a modular processor, with properties similar to Earley's algorithm, by memoizing a simple modular top-down backtracking parser generator. Norvig's memoized parser generator cannot accommodate left-recursive productions but would appear to be as efficient and general as Earley's algorithm in all other respects. According to Norvig, the memoized recognizers have cubic complexity compared to exponential behavior of the original unmemoized versions.

In Norvig's technique, memoization is implemented at the source-code level in Common Lisp through definition and use of a function called `memoize`. When `memoize` is applied to a function `f`, it modifies the global definition of `f` such that the new definition refers to and possibly updates a memo-table. A major advantage of Norvig's approach is that programs may, in some cases, be made more efficient with no change to the source-code definition. In Norvig's approach, both the process of memoizing a function, and the process of updating the memo-table, make use of Common Lisp's updateable function-name space. This precludes direct use of Norvig's approach when language processors are to be constructed in a purely functional programming language where updateable objects are not permitted.

Leermakers [12] and Augusteijn [1] have also described how memoization can be used to improve the complexity of functional top-down backtracking language processors but have not indicated how the memoization process itself would be achieved. In particular, they have not addressed the question of how memoization would be achieved in a purely functional implementation of the language processors.

3. Memoization in purely functional languages

A functional programming language is one in which functions are first-class objects and may, for example, be put in lists, passed to other functions as arguments, and returned by functions as results. A purely functional language, such as Miranda [17], LML, and Haskell [8], is one in which functions provide the only control structure and side-effects, such as assignment, are not allowed. This restriction is a necessary condition for referential transparency, a property of programs that simplifies reasoning about them and which is one of the major advantages of the purely functional programming style [19].

Owing to the fact that side-effects are forbidden, purely functional languages do not accommodate any form of updateable object. Consequently, Norvig's technique for improving the efficiency of top-down backtracking language processors cannot be implemented directly in any purely functional language. However, we can adapt Norvig's approach if we use a variation of memoization that has been described by Field and

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Fig. 3. Definition of the Fibonacci function.

Harrison [5] and investigated in detail by Khoshnevisan [11]. This memoization technique differs from conventional approaches in that memo-tables are associated with the inputs to and outputs from functions, rather than with the functions themselves. A function may be memoized by modifying its definition to accept a table as part of its input, to refer to this table before computing a result, and to update the table before returning it as part of the output. The memo-table is passed as an input to the top-level call of recursively defined functions and is threaded through all recursive calls. To illustrate this technique, we show how the Fibonacci function can be memoized. We begin with a textbook definition given in Fig. 3.

Defined in this way, the evaluation of the Fibonacci function has exponential complexity. The cause of the exponential behavior is the replication of computation in the two recursive calls. This replication can be avoided by memoization. We begin by modifying the definition of `fib` so that it accepts a table as part of its input and returns a table as part of its result. In the modified definition, round brackets and commas are used to denote tuples. The table `t1`, which is output from the first recursive call of `tfib`, is passed as input to the second recursive call of `tfib`. The table `t2`, which is output from the second recursive call, is returned as result from the top-level call of `tfib`:

```
tfib (0, t) = (1, t)
tfib (1, t) = (1, t)
tfib (n, t) = (r1 + r2, t2)
                where
              (r1, t1) = tfib (n - 1, t)
              (r2, t2) = tfib (n - 2, t1)
```

Note that `tfib` still has exponential behavior. When applied to an input, it returns the table unchanged. Rather than modifying the definition of `tfib` directly to make use of the memo-table, as is done in Field and Harrison and in Khoshnevisan, we choose to abstract the table lookup and update process into a general-purpose higher-order function `memo` which we can apply to `tfib` to obtain a memoized version. This variation is comparable to Norvig's technique. When `memo` is applied to a function `f` it returns a new function `newf` whose behavior is exactly the same as `f` except that it refers to, and possibly updates, the memo-table given in the input.

In the definition of the function `memo` below, the expression `mr $pos 1` denotes the first element of the list of memorized results `mr`. The definition of `lookup` makes use of a list comprehension, `[r|(y, r) <- t; y=i]`, which is to be read as "the list of all

```
mfib  (0, t)   = (1,t)
mfib  (1, t)   = (1,t)
mfib  (n, t)   = (r1 + r2, t2)
                  where
                  (r1, t1) = memo  mfib (n - 1, t)
                  (r2, t2) = memo  mfib (n - 2, t1)

mfib (4, []) => (5,[(3,3),(2,2),(0,1),(1,1)])
```

Fig. 4. A memoized version of the Fibonacci function.

r such that the pair (y, r) is a member of the table t and y is equal to the index i."

```
memo  f = newf
          where
          newf  (i, t)  = (r1,t1)
                          where
                          (r1,t1) = (mr $pos 1, t)      ,if mr ~= []
                                  = (r2,update i r2 t2) ,if mr = []
                          (r2,t2) = f (i, t)
                           mr     = lookup i t

update  i r t = (i, r): t
lookup  i t   = [r |(y, r) <- t;  y = i]
```

We can now complete the process of memoizing the Fibonacci function by applying memo to the two recursive calls in the definition of tfib as shown in Fig. 4. The result is a function called mfib which has linear complexity.

Some readers may realize that it is only necessary to store the two most recently computed values of the Fibonacci sequence in the memo-table. Modifying the function update accordingly would decrease the space requirements of mfib but would improve neither time nor space complexity. It should also be noted that there are many other ways to improve the complexity of the Fibonacci function. We do not claim that the use of memoization is the most appropriate technique in this application. We have chosen to use the Fibonacci function as an example so that our technique can be easily compared with that described by Norvig who also used the Fibonacci example for expository purposes.

The technique described above is not as elegant as Norvig's in the sense that the process of memoization has resulted in changes to the definition of the Fibonacci function at the source-code level. Later we show how to reduce the number of changes required for memoization and limit them to local changes only.

4. Memoizing purely functional recognizers

A memoized functional recognizer is a function that takes, as an extra parameter, a memo-table containing all previously computed results. One approach to memoization

is to modify the definitions of the functions `term`, `$orelse` and `$then`, so that the recognizers built with them accept a memo-table as part of their input and return a memo-table as part of their output. Next, a higher-order function `memoize` is applied to each recognizer to create a memoized version of it.

4.1. The memo-table

In order to improve efficiency we have chosen to store the input sequence of tokens in the memo-table and to represent the points at which a recognizer is to begin processing by a list of numbers which are used as indexes into that sequence.

The memo-table is structured as a list of triples of length `n + 1`, where `n` is the length of the input sequence:

```
memo_table == [(num, token, [(rec_name, [num])])]

rec_name == [char]
```

The last element of the memo-table is a special token `#` representing the end of the input. The first component of the `i`th triple is an integer `i`. This number acts as an index into memo-table entries. The second component is the `i`th token in the input sequence. The third component is a list of pairs representing all successful recognitions of the input sequence starting at position `i`. The first component of each pair is a recognizer name, the second component is a list of integer numbers. The presence of a number `j`, where $i \leqslant j \leqslant n + 1$ in this list indicates that the recognizer succeeded when applied to the input sequence beginning at position `i` and finishing at position `j - 1`.

Initially, the third component of each triple in the memo-table is an empty list. The following example shows the initial table corresponding to the input ''aaa''.

```
[(1, 'a', []), (2, 'a', []), (3, 'a', []), (4, '#', [])]
```

Two operations are required for table lookup and update. The operation `lookup` applied to an index `i`, a recognizer name `name` and a memo-table `t` returns a list of previously computed end positions where the recognizer `name` succeeded in processing the input beginning at position `i`. The operation `update` applied to an index `i`, a result `res` and a memo-table `t`, returns a new memo-table with the `i`th entry updated. A result is a pair consisting of a recognizer name and a list of successful end-positions. `Update` adds the result `res` to the list of successful recognitions corresponding to the `i`th token.

```
lookup i name t = []                                        , if i > #t
                = [bs | (x, bs)< - third (t $pos i);
                                                x = name], otherwise
                  where
                  third (x, y, z) = z
```

```
update i res t = map (add_res i res) t
                 where
                 add_res i res (x,term,res_list)
                                  = (x,term,res:res_list), if x = i
                                 = (x, term, res_list)    ,otherwise
```

The function memoize takes as input a recognizer name n, a recognizer f, a list of positions where the recognizer should begin processing the input, and a memo-table. For each start position in the list, the function memoize first calls the function lookup to determine if this application of the recognizer has been computed previously. If lookup returns an empty list, the recognizer is applied, a new result is calculated and the function update is used to add the result to the memo-table. Otherwise the previously computed result is returned. Results returned for each of the start positions are merged with the removal of duplicates.

```
memoize n f ([], t)   = ([], t)
memoize n f (b:bs, t) = (merge_res r1 rs, trs)
                        where
                        (r1, t1) = (mr $pos 1, t), if mr ~= []
                                 = (r2,update b (n,r2) t2), otherwise
                        (r2, t2) = f ([b], t)
                        (rs, trs) = memoize n f (bs, t1)
                        mr  = lookup b n t

merge_res x [] = x
merge_res [] y = y
merge_res (x:xs) (y:ys) = x:merge_res xs    (y:ys), if x < y
                        = y:merge_res (x:xs) ys   , if y < x
                        = x:merge_res xs     ys   , if x = y
```

4.2. The memoized recognizers

The definitions of term, $then and $orelse given in Section 1 are modified to take as input a list of positions where the recognizer should begin processing the input, and a memo-table. Owing to the fact that the entire input sequence of tokens is represented in the memo-table, there is no need for the recognizers to explicitly return unprocessed segments of the input. Instead they return a number as index into the input sequence.

The next modification to the definitions of $orelse and $then is to allow threading of the memo-table through recursive calls. The function term is modified owing to the fact that the input sequence is now stored in the memo-table. The function merge is used to combine and remove duplicates that arise if the same segment of the input can be recognized in more than one way by a recognizer. For recognition purposes such duplicates can be considered equal.

```
The original recognizer                    The memoized version

s = (a $then s $then s)          ms  = memoize "ms"  ((ma $m_then ms $m_then ms)
    $orelse empty                                      $m_orelse empty)

a = term 'a'                     ma  = mterm 'a'
```

Fig. 5. The relationship between a recognizer and its memoized version.

```
mterm c (bs,t)
   = ((concat . (map test_for_c1)) bs, t)
     where
     test_for_c1 b = []      , if b  > #t
     test_for_c1 b = []      , if second (t $pos b) ~= c
     test_for_c1 b = [b + 1], if second (t $pos b)  = c

second (x, y, z) = y

(p $m_orelse q) (bs, t) = (merge_res rp rq, tq)
                            where
                            (rp, tp) = p (bs, t)
                            (rq, tq) = q (bs, tp)

(p $m_then q) (bs, t) = q (rp, tp) , if rp ~= []
                      = ([], tp)    , otherwise
                        where
                        (rp, tp) = p (bs, t)
```

These functions can now be used to improve the complexity of functional recognizers whilst preserving their structural simplicity and modularity. As example, Fig. 5 shows the relationship between the original recognizer for the grammar `s ::= 'a' s s | empty` and the memoized version. Note that it is not necessary to change the definition of `empty` nor is it necessary to memoize the recognizers constructed with `mterm`.

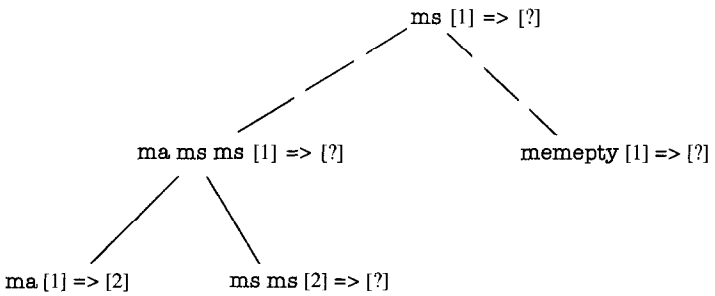
4.3. The algorithm

We begin our description of the algorithm by presenting an example. Suppose that the string ''aa'' is to be processed using the memoized recognizer `ms` defined in Fig. 5. The initial input is as follows, where the second component of the tuple is the initial memo-table:

```
([1], [(1, 'a', []), (2, 'a', []), (3, '#', [])])
```

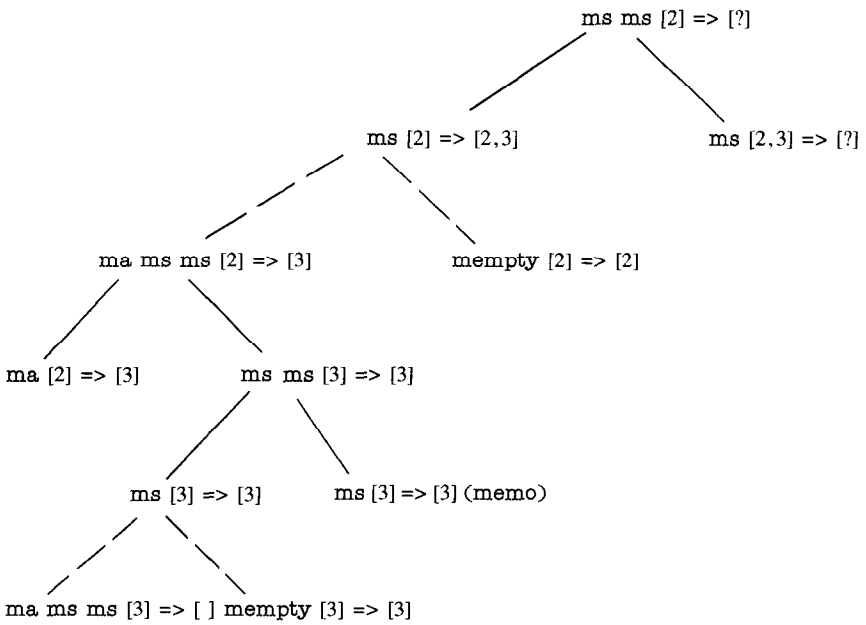
Owing to the fact that no results have been computed yet and that `ms` is an alternation (`$m_orelse`) of two recognizers, the first alternative of `ms` is applied to the initial input.

This recognizer is itself a sequence of the recognizers `ma` and `ms $then ms`, therefore the first of this sequence, i.e. `ma`, is applied to the initial input. The recognizer `ma` succeeds in recognizing an `'a'` and returns a result consisting of a pair with first element `[2]`, indicating that the first element of the sequence of tokens has been consumed, and the memo-table unchanged (because basic recognizers do not update the memo-table). The evaluation tree at this point is as follows, where ? indicates values yet to be computed. Sequencing is denoted by continuous lines and alternation by broken lines.



Next, `ms $then ms` is applied to this result. The application of the first `ms` in this sequence results in a similar computation to the initial application except that the starting position is `[2]`. The same holds when `ms` is applied to position `[3]`.

The third element of the input memo-table corresponds to the end-of-input. The recognizer `ma` applied at position `[3]` fails, returning an empty list, and thus `ma ms ms` fails. The recognizer `mempty` applied at the same position returns as result a tuple whose first element is the list `[3]`. Now the results of both alternatives of the recognizer `ms` have been determined and the value of `ms` applied at position `[3]` is computed. The following shows the evaluation tree when all values up to `ms [2]` have been computed:

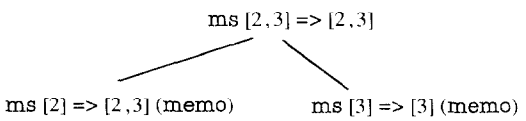


The memo-table is now:

```
[(1,'a',[]),(2,'a',[(''ms'',[2,3])]),(3, '#'. [(''ms'',[3])])]
```

Note that when the recognizer ms is applied to the position [3] for the second time, the corresponding result is simply copied from the memo-table.

When a recognizer is applied to a list that contains more than one element, the result is obtained by applying the recognizer to each element in the list and merging the results. This is illustrated below:



The final result is:

```
([1, 2, 3], [(1, 'a', [(''ms'', [1,2,3])]),
             (2, 'a', [(''ms'', [2,3]  )]),
             (3, '#', [(''ms'', [3]    )])])
```

The following is a more formal description of the algorithm:

1. Input:
   a. A context-free, non-left-recursive, grammar with productions and terminals represented using functions `mterm`, `$m_then`, `$m_orelse`, and `mempty`. The start symbol for the grammar is the name of the first recognizer to be applied.
   b. A pair whose first component is the list `[1]`, and whose second component is a memo-table corresponding to the input sequence of tokens.
2. Output:
   a. A pair whose first component is a list of positions where the recognition process of the input sequence of tokens (starting from the first token) was successfully completed. The second component is the final state of the memo-table.
3. Method:
   a. At each step we apply a recognizer to a list of start positions and a memo-table:
   - If the list is empty, the result is an empty list and the unchanged memo-table.
   - Otherwise, we first apply the recognizer to the first element of the list and the memo-table. The result is a list `r1` and a possibly modified memo-table `t1`. Then we apply the recognizer to the rest of the list and the memo-table `t1`. The result of this application is a list `r2` and a memo-table `t2`. The final result is a pair: a list obtained by merging `r1` and `r2`, and the table `t2`.

b. Application of a recognizer `m` at a position `j` begins by reference to the current memo-table:
   - If the `j`th row of the memo-table contains a result corresponding to `m`, this result is returned.
   - Otherwise a new result is computed, the memo-table is updated and the result returned.

c. Each recognizer can be either the basic recognizer `mempty`, a basic recognizer constructed using `mterm`, or it can be a combination constructed from two or more components using `$m_then` or `$m_orelse`.
   - Results for basic recognizers are obtained immediately by applying the corresponding function.
   - For sequences or alternations, the results of the components are computed first and then combined to obtain the final result.

5. Complexity analysis

We now show that memoized recognizers have a worst-case time complexity of $O(n^3)$ compared to exponential behavior of the unmemoized form. The analysis is concerned only with the variation of time with the length of the input list of tokens. Although a grammar could be very complex, its size will always be independent of the length of the input.

5.1. Elementary operations

We assume that the following operations require a constant amount of time:
1. Testing if two values are equal, less than, etc.
2. Extracting the value of a component of a tuple.
3. Adding an element to the front of a list.
4. Obtaining the value of the `ith` element of a list whose length depends upon the size of the grammar but not on the size of the input list.

5.2. The size of the memo-table

The memo-table is structured as a list of `(n + 1)` tuples, where `n` is the length of the input sequence of tokens. The first component of each tuple is an integer ranging from `1` to `n + 1`. The second component of a tuple whose first component is `i`, is the `ith` token in the input. The third component is a list of pairs `(recognizer_name, result)`. Owing to the fact that the grammar is fixed, the number of recognizers, denoted by `r`, is constant. Therefore, for each tuple in the memo-table, the length of the list of pairs is $\leqslant$ `r`.

The second component of each pair is a list of positions represented by integers where the corresponding recognizer succeeded in completing the recognition of a segment of the input. The length of the lists that correspond to the `ith` tuple is at most `(n - i + 2)` owing to the fact that a recognizer applied to input at position `i` may succeed at any position `j`, $i \leqslant j \leqslant n + 1$.

5.3. Memo-table lookup and update

The function `lookup` applied to an index `i`, a recognizer name, and a memo-table first searches the memo-table to access the `ith` element, then it searches the list of results in the `ith` tuple to access the element that corresponds to the given recognizer name. The function `lookup` requires O(n) time.

The function `update` applied to an index `i`, a result `res`, and a memo-table returns a new memo-table with the `ith` tuple updated. The result `res` is added in front of the list of successful recognitions corresponding to the `ith` token. The function `update` requires O(n) time.

5.4. Basic recognizers

Application of the recognizer `mempty` simply creates a pointer to the input. This takes constant time.

Application of a recognizer `mterm a` to a single start position `i` requires the `ith` entry in the memo-table to be examined to see if the `ith` token is equal to `a`. If there is a match then the result `i + 1` is added to the list of results returned by `mterm`. Otherwise the recognizer fails. This operation is O(n).

Note that we are only considering, here and in the next two subsections, the time required to apply a recognizer to a single position in the input list. We consider application of a recognizer to a more-than-one-element list later.

5.5. Alternation

Assuming that the results `p i` and `q i` have been computed, application of a memoized recognizer (`p $m_orelse q`) to a single start position `[i]` involves the following steps:

- one memo-table lookup – O(n)
- and, if the recognizer has not been applied before:
  - merging of two result lists, each of which is in the worst case of length `n + 1`, – O(n),
  - one memo-table update – O(n).

5.6. Sequencing

Assume that `p [i]` has already been calculated. In the worst case the result is the list `[i, i + 1, . . , n - 1]`. Assume also that `q [i, i + 1, . . , n + 1]` has already been calculated. Now, application of a memoized recognizer `p $m_then q`

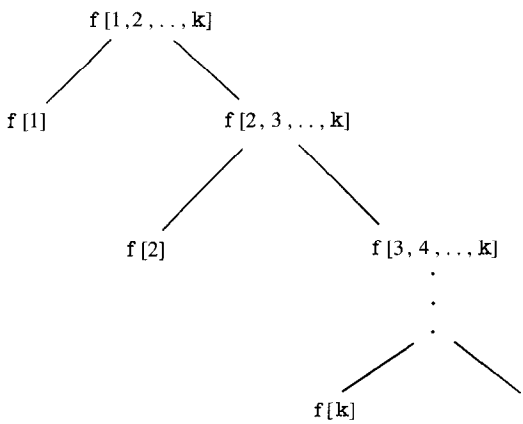
to a single start position i involves the following steps:

- one memo-table lookup (O(n)) and if the lookup fails, computation of the result plus,
  - one memo-table update (O(n)).

5.7. *Merging results returned when a recognizer is applied to a list of start positions*

The function merge is also used to combine the results returned by a single memoized recognizer when applied to a list of start positions with more than one entry (see the definition of memoize).

Suppose a recognizer f is applied to a k-element list of start positions [1, . . ,k]. The corresponding evaluation tree is as follows:



Assuming that the results of f [i] and f [i + 1, . . ,k] have already been computed, computation of f [i, i + 1, . . ,k] requires one memo-table lookup (O(n)) and one merge, of two lists which are in the worst case of length n + 1. The total time is O(n).

Note also that application of a recognizer f to a k-element list of start positions results in an execution tree with 2*k + 1 nodes representing applications of the recognizer f.

5.8. *The execution tree*

The analysis so far can be summarized in terms of execution trees (such as those shown earlier). Each non-leaf node of an execution tree corresponds to an application of a recognizer to a list of start positions, or to an application of m_orelse or m_then. Leaf-nodes correspond either to an application of mempty, or mterm a for some a, or to a computation that has been performed before and stored in the memo-table.

Lemma 1. *We have shown that the result corresponding to* `mempty`, `mterm a` *for some* `a`, *and* `lookup` *can be computed in* O(n) *time.*

Lemma 2. *We have also shown that results corresponding to non-leaf nodes can be computed in* O(n) *time provided that the values of their children are available.*

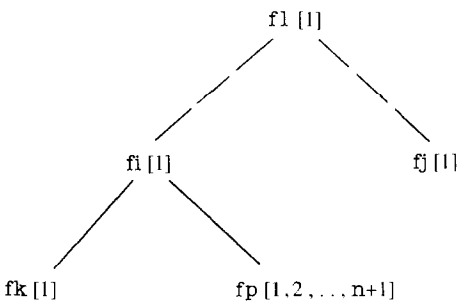
5.9. *Proof of* $O(n^3)$ *time complexity*

Theorem. Given an arbitrary context-free non-left-recursive grammar G, the corresponding memoized functional recognizer requires $O(n^3)$ time to process an input sequence of length n. If the grammar is not ambiguous, the time complexity is $O(n^2)$

Proof. Let $f_1, f_2, \ldots, f_r$ be a set of recognizers corresponding to the grammar G, and let f_1 correspond to the start symbol in the grammar. We begin by applying the recognizer f_1 to the list [1]. This application yields an execution tree similar to the ones shown earlier. We will show that for an arbitrary grammar the number of nodes in such a tree is $O(n^2)$, and if the grammar is not ambiguous this number reduces to O(n). Owing to the fact that the time required to perform computations at each node is linear in the length of the input sequence (Lemmas 1 and 2), this concludes the proof of the theorem.

For simplicity assume that each recognizer is either `mempty`, `mterm a` for some `a`, or is of the form `(p $m_orelse q)`, or `(p $m_then q)` for some `p` and `q`. In practice recognizers can be a combination (`$m_orelse` and/or `$m_then`) of more than two recognizers, but the number will always be bounded by the size of the grammar and will be independent of the length of the input sequence of tokens.

Suppose that the recognizer f_1 is of the form $(f_i$ `$m_orelse` $f_j)$ for some $2 \leqslant i, j \leqslant r$. Suppose also that the recognizer f_i is of the form $(f_k$ `$m_then` $f_p)$ for some $2 \leqslant k$, $p \leqslant r$ and $k \neq i$, $p \neq i$. The corresponding tree in the worst case is as follows:



Consider the expansion of those subtrees that correspond to an application of a recognizer to the one-element list of start positions [1]. Owing to the fact that the grammar is non-left-recursive and that it consists of r recognizers, after a maximum number of steps in each path, which depends only on the size of the grammar, there must be an application of a recognizer that consumes some input. It follows that the

total number of applications of a recognizer to a one-element list is independent of the length of the input. For the same reason, the total number of applications of a recognizer to a more than one-element list (in the worst case an `(n + 1)`-element list) is independent of the length of the input.

When the first step is completed, there will be only O(`r`) subtrees to be further expanded. This is because the result corresponding to a pair (recognizer, start position) is calculated only once. If the same recognizer is applied to the same start position again, the corresponding result is simply copied from the memo-table.

At the next stage, the same procedure is repeated for each recognizer that is applied to the list `[2]`. The only difference is that now O(`r`) subtrees must be expanded not just one. Only O(`r`) nodes will be generated for each recognizer applied to the list `[2]`, and O(`r`) nodes for each application of a recognizer to a more-than-one-element list.

At the `ith` step, there will be O(`r`) nodes corresponding to an application of a recognizer to the list `[i]`, and O(`r`) nodes corresponding to an application of a recognizer to a more-than-one-element list (in the worst case, an `(n - i + 2)`-element list). The total number of steps is `n + 1`.

Owing to the fact that an application of a recognizer to a `k`-element list yields a tree that contains `2 * k + 1` nodes, as discussed in Section 5.7, the total number of nodes is given by the following, where `c` is proportional to the number of recognizers `r`.

$$N = \sum_{k=1}^{n+1}(c + c*(2*k+1)) = 2*c*(n+1) + 2*c*(n+1)*(n+2)/2 = O(n^2).$$

If the grammar is not ambiguous then each input sequence of tokens can be recognized in just one way. Therefore, each recognizer applied at some position `i` will return at most a one-element list as result. The corresponding formula for unambiguous grammars given below concludes the proof.

$$N = \sum_{k=1}^{n+1}(c + c*1) = 2*c*(n+1) = O(n).$$

6. A monadic approach to incorporate memoization

So far, we have used an ad hoc method to redefine the recognizer functions in order to incorporate memoization. This method is susceptible to error. In fact, an earlier version of the paper contained an insidious error: the function `m_then` was defined as follows:

```
(p $m_then q) (bs, t) = q (rp, tp)   , if rp ~= []
                      = ([], t)      , otherwise
                        where
                        (rp, tp) = p (bs, t)}
```

According to this definition, if the recognizer p fails, then the memo-table is returned unchanged in the result `([], t)`. This error would result in exponential complexity for certain grammars when applied to certain inputs which fail to be recognized.

In this section, we show how the recognizer definitions can be modified in a structured way which reduces the possibility of such errors. The method treats memoization as a specific instance of the more general notion of adding features to purely functional programs.

6.1. Monads

Monads were introduced to computing science by Moggi [15] who noticed that reasoning about programs that involve handling of the state, exceptions, I/O, or non-determinism can be simplified, if these features are expressed using monads. Inspired by Moggi's ideas, Wadler [21] proposed monads as a way of structuring functional programs. The main idea behind monads is to distinguish between the type of values and the type of computations that deliver these values. A monad is a triple (`M`, `unit`, `bind`) where `M` is a type constructor, and `unit` and `bind` are two polymorphic functions. `M` can be thought of as a function on types, that maps the type of values into the type of computations producing these values. `unit` is a function that takes a value and returns a corresponding computation; the type of `unit` is `a -> Ma`. The function `bind` represents sequencing of two computations where the value returned by the first computation is made available to the second (and possibly subsequent) computation. The type of `bind` is

```
Ma -> (a -> Mb) -> Mb
```

The identity monad [21] below represents computations as the values they deliver.

```
id * == *

unit1 :: * -> id *
unit1 x = x

bind1 :: id * -> (* -> id **) -> id **
(p $bind1 k) = k p
```

The state monad (also defined in [21]) is an abstraction over computations that deal with the state. The definition is given below:

```
stm * == state -> (*, state)
state == [(num, [([char], [num])])]

unit2 :: * -> stm *
unit2 a = f
          where f t = (a, t)

bind2 :: stm * -> (* -> stm **) -> stm **
```

```
(m $bind2 k) = f
               where
               f x = (b, z)
                      where
                     (b, z) = k a y
                              where
                              (a, y) = m x
```

We will use the identity and the state monad to construct non-memoized and memoized monadic recognizers, respectively. In the description below, we refer to a third monad which we use as an analogy in the construction of our monadic recognizers. This is the monad for lists [21]. Owing to the fact that our recognizers can be applied to a list of inputs, it is necessary to have a well-structured way of doing that.

```
list * == [*]

unit :: * -> list *
unit a = [a]

bind :: list * -> (* -> list **) -> list **
[] $bind y = []
(a:x) $bind y = (y a) ++ (x $bind y)
```

6.2. *Non-memoized monadic recognizers*

In order to use monads to provide a structured method for adding new effects to a functional program, we begin by identifying all functions that will be involved in those effects. We then replace those functions, which can be of any type `a -> b`, by functions of type `a -> Mb`. In effect, we change the program so that selected function applications return a computation on a value rather than the value itself. This computation may be used to add features such as state to the program. In order to effect this change, we use the function `unit` to convert values into computations that return the value but do not contribute to the new effects, and the function `bind` is used to apply a function of type `a -> Mb` to a computation of type `Ma`. Having made these changes, the original program can be obtained by using the identity monad `idm`, as shown below. In order to add new effects such as state, or exceptions, we simply change the monad and make minimal local changes as required to the rest of the program. In the following subsection, we show how to add the new effect of memoization by replacing the identity monad with the state monad `stm`, and making some local changes.

The non-memoized recognizers introduced earlier in this paper were functions taking a list of input sequences of tokens and returning a similar list of sequences yet to be processed. The definition of the non-memoized monadic recognizers differs slightly in that the list of inputs is represented by a pair : a list of start positions and the whole

input sequence of tokens. Owing to the fact that the input sequence remains unchanged during the execution of the program, there is no need for any recognizer to return it.

In order to construct non-memoized monadic recognizers, we start by defining the type of non-memoized recognizers. We define it using the type constructor `idm` of the identity monad.

```
rec * = [char] -> [num] -> idm *
```

That is, a recognizer of type a is a function that applied to an input string and a list of start positions returns an "identity" computation of type a. We can now define the function `term_1` which, when applied to a character, returns a function that is always applied to a one-element list of positions.

```
term_1 :: char -> rec [num]
term_1 c s [x] = unit1 [], if (x > #s) \/ (s!(x-1) ~= c)
               = unit1 [x+1], otherwise
```

In analogy with the bind operator for the list monad, the function `term`, which when applied to a character returns a recognizer that can be applied to more than one-element list of start positions, is defined as follows:

```
term :: char -> rec [num]
term c s [] = unit1 []
term c s (x:xs) = term_1 c s [x] $bind1 f
                  where f a = term c s xs $bind1 g
                              where g b = unit1 (merge_res a b)
```

The definitions of `orelse`, `then` and `empty` are given below. Note that we have replaced the append operator ++ with the function `merge_res` that combines two lists removing duplicates.

```
orelse ::  rec [num] -> rec [num] -> rec [num]
(p $orelse q) s input = p s input $bind1 f
                        where f a = q s input $bind1 g
                        where g b = unit1 (merge_res a b)

then :: rec [num] -> rec [num] -> rec [num]
(p $then q) s input = p s input $bind1 f
                        where f a = unit1 [], if a = []
                                  = q s a   , if a ~= []

empty :: rec [num]
empty s x = unit1 x
```

Notice that we have not rewritten the application of `merge-res` using `bind1`. The reason for this is that we know that in this application, `merge_res` will not be involved

in memoization and therefore the result of its application can be viewed as a value rather than a computation.

6.3. Memoized monadic recognizers

We now consider the state monad as given earlier and define the two operations on the state: lookup and update. The type of the state is [num, [([char, [num])]].

```
lookup :: num -> [char] -> stm [[num]]
lookup ind name st
   = ([], st)                                          , if ind > #st
   = ([bs | (x,bs) <- (snd (st!(ind-1)));x=name], st) , otherwise

update :: num -> [char] -> [num] -> stm ()
update ind name val st
        = (undef, map (update_mt_entry ind name val) st)

update_mt_entry ind name val (x, list)
  = (x, (name, val) : list), if x = ind
  = (x, list)              , otherwise
```

We define the type of the memoized recognizers in terms of the type constructor of the state monad stm:

```
rec * == [char] -> [num] -> stm *
```

and define the function memoize (there is an analogy to the definition of term and bind for the list monad here).

```
memoize :: [char] -> mrec [num] -> mrec [num]
memoize name f s [] = unit2 []
memoize name f s (x:xs) = memoize1 name f s [x] $bind2 g
                          where
                          g a = memoize name f s xs $bind2 h
                                where
                                h b = unit2 (merge_res a b)

memoize1 :: [char] -> mrec [num] -> mrec [num]
memoize1 name f s [i] = lookup i name $bind2 g
                        where
                        g a = unit2 (a!0), if a ~= []
                            = f s [i] $bind2 h, otherwise
                              where
                              h b = update i name b $bind2 r
                                    where
                                    r any = unit2 b
```

Table 1

Number of 'a's	Number of reductions		Space of bytes	
	Unmemoized	Memoized	Unmemoized	Memoized
3	2 132	4 490	3 097	5 473
6	24 914	11 867	35 389	14 407
9	222 792	24 396	315 187	29 841
12	1 830 567	43 638	2 587 678	53 943
15	14 726 637	70 935	20 814 328	88 404
18	117 938 202	107 745	166 686 913	135 168
21	Out of space	155 526	Out of space	196 179

The definitions of `term`, `orelse`, and `then` remain unchanged except that `unit1` and `bind1` are replaced by `unit2` and `bind2`, respectively. The memo-table is completely hidden in the definition of `term`, `orelse`, and `then`. One of the advantages is that having identified all recognizer functions as being involved in the memoization effect, the monadic form of `then` is straightforward and thereby this approach reduces the chance of making the kind of error referred to at the beginning of this section.

The definition of monadic memoized recognizers is exactly the same as with the original memoized recognizers, and the complexity analysis presented earlier holds also for memoized monadic recognizers.

Table 1 shows the results obtained when an unmemoized monadic recognizer for the grammar `s ::= a s s | empty` and a memoized monadic version were applied to inputs of various length. The results suggests that the recognizers also have $O(n^3)$ space complexity as well as $O(n^3)$ time complexity.

More information on the use of monads to structure functional language processors can be found in Wadler [20–22].

7. Memoizing parsers and syntax-directed evaluators

The memoization technique presented in this paper could be readily extended so that the memo-tables contain parse tables similar to those created by Earley's algorithm [3], or the more compact representation of factored syntax trees suggested by Leiss [13]. However, to do so would not be in keeping with an approach that is commonly used by the purely functional programming community in building language processors. That approach is to avoid the explicit construction of syntax trees unless the trees are specifically required to be displayed as part of the output. Instead of constructing labelled syntax trees which are subsequently evaluated, an alternative approach is used: semantic actions are closely associated with the executable grammar productions so that semantic attributes are computed directly without the need for the explicit representation of syntax trees. User-defined types can be introduced to accommodate different types

of attributes as has been done in the W/AGE attribute grammar programming language [7]. This approach is viable owing to the lazy-evaluation strategy employed by most purely functional languages. The memoization technique described above can be used to improve the efficiency of such syntax-directed evaluators with two minor modifications:

1. The definition of `m_orelse` is changed so that the function `merge` is replaced with a function that removes results that are regarded as duplicates under application-dependent criteria which may be less inclusive than the criterion used for recognizers. Results that are returned by a recognizer are regarded as duplicates if they have the same end points. For recognition purposes the end points are all that is required to be maintained in the memo-table. With syntax-directed evaluators, the end points may be augmented with semantic values. A single end point pair may have more than one value associated with it. In some cases syntactic ambiguity may result in semantic ambiguity. Results returned by a language processor would only be regarded as being duplicates if they have the same end points and have equivalent semantic attributes. The function `merge` would be replaced by an application-dependent function that identifies and removes such duplication. In this approach, if syntax trees are required as part of the output, they are simply treated as another attribute. In such cases syntactic ambiguity is isomorphic with semantic ambiguity and the function `merge` would be replaced by concatenation in the definition of `m_orelse`.
2. The memo-tables and the update and lookup functions are modified according to the attributes that are required in the application.

One advantage that derives from this approach is that all unnecessary computation is avoided. Memoization prevents language processors from reprocessing segments of the input already visited and the use of `merge`, or an application-dependent version of it, removes duplication in subcomponents of the result as soon as it is possible to detect it. It should be noted that the complexity of language processors constructed in this way is application dependent. If syntax trees are required to be represented in full, the language processor may have exponential complexity in the worst case owing to the fact that the number of syntax trees can be exponential in the length of the input for highly ambiguous grammars. A compact representation of the trees could be produced in polynomial time and the trees could then be passed on to an evaluator. However, this would detract from the modularity of the language processor and would provide no benefit if the trees were to be subsequently displayed or otherwise processed separately as this could be an exponential process.

8. Concluding comments

This paper was inspired by Norvig's demonstration that memoization can be implemented at the source-code level in languages such as Common Lisp to improve the efficiency of simple language processors without compromising their simplicity. We have shown that Norvig's technique can be adapted for use in purely functional programming languages that do not admit any form of updateable object. The technique

described in this paper can be thought of as complementing that of Norvig's in that it enables memoization to be used to improve the efficiency of highly modular language processors constructed in purely functional languages.

This application has also illustrated how monads can be used to structure functional programs in order to avoid errors when modifications such as the addition of state are made. We are now exploring the use of monads in the memoization of programs that are constructed as executable attribute grammars.

Acknowledgements

We would like to thank Young Gil Park, Dimitris Phoukas, and Peter Tsin of the University of Windsor for helpful discussions, and the anonymous referees for a careful reading of the paper, for many useful suggestions, and for identifying the error in the code that was discussed in Section 6. Richard Frost also acknowledges the support received from the Canadian Natural Sciences and Engineering Research Council.

References

[1] L. Augusteijn, *Functional Programming, Program Transformations and Compiler Construction* (Philips Research Laboratories, 1993).

[2] W.H. Burge, *Recursive Programming Techniques* (Addison-Wesley, Reading, MA, 1975).

[3] J. Earley, An efficient context-free parsing algorithm, *Commun. ACM* **13** (2) (1970) 94–102.

[4] J. Fairburn, Making form follow function: An exercise in functional programming style, University of Cambridge Computer Laboratory Technical Report No. 89, 1986.

[5] A.J. Field and P.G. Harrison, *Functional Programming* (Addison-Wesley, Reading, MA, 1988).

[6] R.A. Frost, Constructing programs as executable attribute grammars, *Comput. J.* **35** (1992) 376–389.

[7] R.A. Frost, W/AGE The Windsor Attribute Grammar Programming Environment, *Schloss Dagstuhl International Workshop on Functional Programming in the Real World*, 1995.

[8] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, S. Peyton Jones, M. Reeve, D. Wise and J. Young, Report on the programming language Haskell, a non-strict, purely functional language, Version 1.2, *ACM SIGPLAN Notices* **27** (5) (1992).

[9] R.J.M. Hughes, Lazy memo functions, in: G. Goos and J. Hartmanis, eds., Proc. *Conf. on Functional Programming and Computer Architecture*, Nancy, France, September 1985, Springer Lecture Note Series, Vol. 201 (Springer, Berlin, 1985).

[10] G. Hutton, Higher-order functions for parsing, *J. Functional Programming* **2**(3) (1992) 323–343.

[11] H. Khoshnevisan, Efficient memo-table management strategies, *Acta Inform.* **28** (1990) 43–81.

[12] R. Leermakers, *The Functional Treatment of Parsing* (Kluwer Academic Publishers, Dordrecht, 1993).

[13] H. Leiss, On Kilbury's modification of Earley's algorithm, *ACM TOPLAS* **12** (1990) 610–640.

[14] D. Michie, 'Memo' functions and machine learning, *Nature* **218** (1968) 19–22.

[15] E. Moggi, Computational lambda-calculus and monads, *IEEE Symp. on Logic in Computer Science*, Asilomar, Ca (June 1989) 14–23.

[16] P. Norvig, Techniques for automatic memoisation with applications to context-free parsing, *Computational Linguistics* 17 (1) (1991) 91–98.

[17] D. Turner, A lazy functional programming language with polymorphic types, *Proc. IFIP Internat. Conf. on Functional Programmiong Languages and Computer Architecture*, Nancy, France, Springer Lecture Notes in Computer Science, Vol. 201 (Springer, Berlin, 1985).

[18] P. Wadler, How to replace failure by a list of successes, in: P. Jouannaud, ed. *Functional Programming Languages and Computer Architectures*, Lecture Notes in Computer Science, Vol. 201 (Springer, Heidelberg, 1985) 113.

[19] P. Wadler, ed., Special issue on lazy functional programming, *Comput. J.* 32 (2) (1989).

[20] P. Wadler, Comprehending monads, in: *ACM SIGPLAN/SIGACT/SIGART Symp. on Lisp and Functional Programming*, Nice, France (June 1990) 61–78.

[21] P. Wadler, *Monads for Functional Programming*, Marktoberdorf Summer School on Program Design Calculi, Springer Lecture Notes in Computer Science, 1992.

[22] P. Wadler, Monads for functional programming, in: J. Jeuring and E. Meijer eds., *Proc. Bastad Spring School on Advanced Functional Programming*, Springer Lecture Notes in Computer Science, Vol. 925 (Springer, Berlin, 1995).